

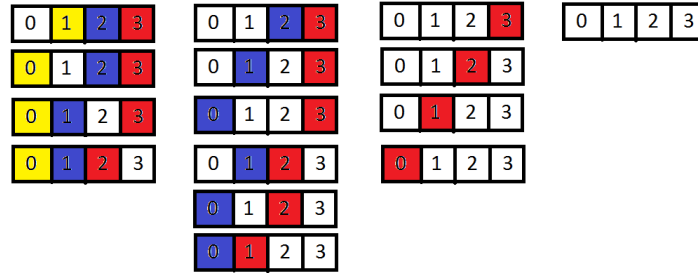
ON THE GENERATION OF BOUNDARY MATRICES FOR SIMPLICIAL AND DELTA COMPLEXES

MARIAN ANTON AND LANDON RENZULLO

ABSTRACT. Topological data analysis studies point-clouds in high dimensional spaces by generating simplicial complexes and calculating their homology. For example, the subsets of place neurons that co-fire generate a simplicial complex having the topology of the environment. See [2]. In the literature simplicial complexes are generated by listing all the faces of each simplex and by removing the duplicates. See [3]. Our algorithm does not produce duplicates and can also generate Δ -complexes and their boundary maps. See [1]. To achieve this we make use of the divide and conquer paradigm and dynamic programming based on statistical hash tables.

1. DIVIDE AND CONQUER PARADIGM

This is a strategy to solve a problem by splitting it into multiple smaller problems and solve them recursively. For example, we get all the *faces* of a *simplex* $s = [0, 1, 2, 3]$ by recursively deleting one *vertex* at a time as follows:



In Python, for each natural number p the following function generates faces by removing only vertices below the position $\text{len}(s) - p$ in the simplex s and $\text{com}(s, 0) + [s]$ is the *simplicial complex* generated by s .

Date: January 4th, 2017.

```

def com(s,p):
    if len(s)==1:
        return []
    else:
        l=[]
        for i in range(p,len(s)):
            n=len(s)-i-1
            ss=s.copy()
            ss.remove(s[n])
            l.append(ss)
            l=l+com(ss,i)
        return l

```

The complex recursively generated by $s = [0, 1, 2, 3]$ looks like this:

```

>>> import tda
>>> s=[0,1,2,3]
>>> tda.com(s,0)+[s]
[[0, 1, 2], [0, 1], [0], [1], [0, 2], [2], [1, 2], [0, 1, 3], [0, 3], [3],
[1, 3], [0, 2, 3], [2, 3], [1, 2, 3], [0, 1, 2, 3]]

```

To make it usable we lexicographically order this complex with respect to $\dim(t) = \text{len}(t) - 1$ and the natural ordering of vertices by a quick sort algorithm where t runs over all faces of s .

```

>>> tda.Lx(tda.com(s,0)+[s])
[[0], [1], [2], [3], [0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3], [0, 1, 2],
[0, 1, 3], [0, 2, 3], [1, 2, 3], [0, 1, 2, 3]]

```

2. STATISTICAL HASH TABLES

A *hash* function h pseudo-randomly assigns to each entry s in a data-set an integer $k = h(s)$ that can be used to quickly reference s . Its fibers $h^{-1}(k)$ are almost uniformly distributed for naturally occurring data-sets. For example, the hash function h could be injective on the faces of the simplex $s = [0, 1, 2, 3]$ and the integers $8 = h(t)$ and $24 = h(u)$ are references for the faces $t = [2, 3]$ and $u = [0, 1, 3]$. We define a *hash table* to be essentially the list H of fibers $H[k] = h^{-1}(k)$ of a hash function h over the set of integers k and use it to locate each entry s in the data-set by finding the fiber $h^{-1}(k)$ containing s and then searching for s inside that fiber.

Dynamic programming is a strategy which uses recursion to solve a problem once and *memoization* to store the solution in a hash table for future use. For example, in the hash table of the simplex $s = [0, 1, 2, 3]$ the face $t = [2, 3]$ located in fiber $H[8]$ is lexicographically the 10th

among all faces and the 6th among all edges; the face $u = [0, 1, 3]$ in the fiber $H[24]$ is the 12th among all faces and the 2nd among all triangles. To retrieve this information about u we do $h(u) = 8$ and search for u in the fiber $H[8]$ instead of searching for u in the whole list of faces.

```
>>> tda.H
[0, 0, 0, [[4, 0, [0, 1]], [5, 1, [0, 2]]], [0, 0, [0]], 0, 0, 0, [9, 5, [2, 3]], [[13, 3,
[1, 2, 3]], [14, 0, [0, 1, 2, 3]]], 0, 0, [[1, 1, [1]], [3, 3, [3]]], 0, 0, 0, 0, 0, [6,
2, [0, 3]], 0, [12, 2, [0, 2, 3]], 0, 0, [[2, 2, [2]], [7, 3, [1, 2]], [11, 1, [0, 1, 3]]]
, [[8, 4, [1, 3]], [10, 0, [0, 1, 2]]], 0, 0, 0, 0]
>>> tda.H[8]
[9, 5, [2, 3]]
>>> tda.H[24]
[[2, 2, [2]], [7, 3, [1, 2]], [11, 1, [0, 1, 3]]]
```

3. THE BOUNDARY MAPS

For a disjoint union U of simplices, we generate their faces without duplicates, lexicographically order them in a complex $C = C(U)$ and encode this information in a hash table H for a quick retrieval.

The *boundary map* of the complex C is a matrix D with the only non-zero entries $(-1)^i$ in column s and row s_i , the face of s given by removing the i -th vertex, where $i = 0, 1, \dots, \dim(s)$ and s runs over all simplices of C . We construct each s -column of the block $D[k]$ of D corresponding to $\dim(s) = k$ by locating the faces s_i of s using the hash table H . The output is the boundary maps for the homology of the space U .

The complex $C = C(U)/R$ could be a Δ -complex generated by a disjoint union U of simplices with identifications R . For example, the *torus* can be generated by two triangles (JPlex requires 18 triangles) U : $[0, 1, 2]$ and $[3, 4, 5]$ with three *identifications*

$$R: [1, 2] = [3, 4], [0, 1] = [4, 5], [0, 2] = [3, 5].$$

These induce *face-identifications* $[1] = [3]$, $[2] = [4]$, $[0] = [4]$, etc. that are encoded in the hash table H for $C(U)$. The boundary maps D for C are obtained in a similar way as the boundary maps for $C(U)$ but making updates based on R .

We initialize D as a properly sized zero matrix and successively update its entries for s' a face of a simplex s in U and p a natural number:

```
def B(s,p):
    k = len(s) - 1
    a = position of s in C(U) relative to k
    for 0 < q < len(s):
        remove vertex s[k - q] and produce a face s'
```

$b = \text{position of } \text{lor}(s') \text{ in } C(U) \text{ relative to } k - 1$
 if $k - q$ is even: $D[k][b, a] = D[k][b, a] + 1$
 else: $D[k][b, a] = D[k][b, a] - 1$
 if $q \geq p : B(\text{lor}(s'), q)$

Here $\text{lor}(s')$ denotes the lexicographically lowest simplex in $C(U)$ that is R -identified with the face s' . The boundary map D for the complex C is given by $B(s, 0)$ for all s in U .

For the torus, our algorithm produces the boundary map D . This gives the boundary map below for the Δ -structure after removing simplices which are not updated.

```

>>> import tda2
>>> tda2.tor
[[[0, 2], [3, 5]], [[0, 1], [4, 5]], [[1, 2], [3, 4]]]
>>> tda2.setGenData([[2, 2]], tda2.tor)
>>> tda2.D
[Matrix([[0]]), Matrix([[0, 0, 0]]), Matrix([
[ 1,  1,
[-1, -1,
[ 1,  1]])]

```

4. THE ALGORITHM

```

def complexify(s, p = 0): #generates the complex of simplex s when p is left as 0.
    Otherwise a subcomplex of faces with vertices < p never removed, is generated
    if len(s)==1:
        return [s]
    else:
        l=[]
        for i in range(p, len(s)):
            n=len(s)-i-1
            ss=s.copy()
            ss.remove(s[n])
            l.append(ss)
            l=l+complexify(ss, i)
        return l + [s]

def isLess(s, t): #lexicographical ordering boolean function for simplices.
    if s < t, returns true
        n=min(len(s), len(t))
        for i in range(0, n):
            if s[i]<t[i]:
                return True
            elif s[i]>t[i]:
                return False
        if len(s)<len(t):
            return True
        return False

```

```

import random, BettiCalc

def MakeLex(l): #quicksort implimented with lexicographic ordering
that also preferences size of simplices
    if len(l)<=1:
        return l
    p=random.randrange(0,len(l))
    piv=l[p]
    a=[]
    b=[]
    for i in range(0,len(l)):
        y = [len(l[i])+l[i]
        z =[len(piv)]+piv
        if isLess(y,z) and i!=p:
            a.append(l[i])
        if isLess(z,y) and i!=p:
            b.append(l[i])
    l=MakeLex(a)+[piv]+MakeLex(b)
    return l

#import sympy

from sympy import *

def choose(n,k): #standard combination function
    if 0<=k<=n:
        num=1
        den=1
        for t in range(1,min(k,n-k)+1):
            num*=n
            den*=t
            n-=1
        return num//den
    else:
        return 0

def j(gen,n): #returns the number of n dimensional simplices that will be
generated from the generating set gen
    sum=0
    for i in range(0,len(gen)):
        sum=sum+gen[i][0]*choose(gen[i][1]+1,n+1)
    return sum

def par(gen): #returns partition generated from generating set gen
    P=[]
    N=[]
    for i in range(0,j(gen,0)):

```

```

        N.append(i)
    for t in range(0,len(gen)):
        n=gen[t]
        for k in range(0,n[0]):
            P.append(N[0:n[1]+1])
            N=N[n[1]+1:len(N)]
    return P

def M(gen): #generates the full master list of simplices from gen
    P=par(gen)
    l=[]
    for i in range(0,len(P)):
        l=l+complexify(P[i],0)
    return MakeLex(l)

def h(s): #converts a simplex to string and returns its hash value
    str1 = ','.join(str(e) for e in s)
    return hash(str1)

def location(s,x = 'r'): #uses hash table H to find quickly find the relative ('r')
or absolute ordering of a simplex
    k = h(s)%len(H)
    if type(H[k][0]) == list:
        for i in range(0,len(H[k])):
            if H[k][i][2] == s:
                if x == 'r':
                    return H[k][i][0]
                else:
                    return H[k][i][1]
    else:
        if x == 'r':
            return H[k][0]
        else:
            return H[k][1]

def setGenData(s,r = []):
    global master,D,H,deadsimps
    master.clear()
    D.clear()
    deadsimps.clear()
    master = M(s)
    D=[zeros(1,j(s,0))]
    d = s[len(s)-1][1]
    for i in range(0,d): #initializes boundary matrix D
        D.append(zeros(j(s,i),j(s,i+1)))
    H.clear()

```

```

for i in range(0,2*len(master)): #initializes ordinal hash table H
    H.append(0)
c = 0
for i in range(0,len(master)): #fills H with simplex positions
    n = h(master[i])%len(H)
    if len(master[i]) > len(master[i-1]):
        c = 0
    if H[n] == 0:
        H[n] = [c,i,master[i],master[i]]
    elif type(H[n][0]) == list:
        H[n].append([c,i,master[i],master[i]])
    else:
        H[n] = [H[n],[c,i,master[i],master[i]]]
    c = c+1

initializeRelations(r)
makeBoundary(s)
pruneSimps()

global D,H,master,tor,kln,deadsimps
master = []
D = []
H = []
deadsimps = []
tor = [[[0,2],[3,5]],[[0,1],[4,5]],[[1,2],[3,4]]]
kln = [[[0,2],[3,5]],[[0,1],[4,5]],[[1,2],[3,4]]]

def boundarydecomp(s,p = 0): #takes simplex s, and recursively calls a rightbound.
    Will update all boundaries of the COMPLEX of s (s and all faces)
    global D,deadsimps
    if type(s) == int:
        s = [s]
    dim = len(s) - 1
    m = D[dim]
    for x in range(0,p):
        if len(s) <= 1:
            break
        ss = s.copy()
        ss.remove(s[dim-x])
        pos = location(lowestOrderRelation(ss),'r')
        if lowestOrderRelation(ss) != ss:
            deadsimps.append(ss)
        if (dim-x)%2 == 0:
            m[pos,location(s,'r')] = m[pos,location(s,'r')] + 1
        else:
            m[pos,location(s,'r')] = m[pos,location(s,'r')] - 1
    for y in range(p,len(s)):

```

```

    if len(s) <= 1:
        break
    ss = s.copy()
    ss.remove(s[dim-y])
    pos = location(lowestOrderRelation(ss), 'r')
    if lowestOrderRelation(ss) != ss:
        deadsimps.append(ss)
    if (dim-y)%2 == 0:
        m[pos,location(s, 'r')] = m[pos,location(s, 'r')] + 1
    else:
        m[pos,location(s, 'r')] = m[pos,location(s, 'r')] - 1
    boundarydecomp(lowestOrderRelation(ss), y)

def makeBoundary(s):
    P = par(s)
    for i in range(0, len(P)):
        boundarydecomp(P[i], 0)

def initializeRelations(Rel): #gets the full set of relations, and updates H with
relation information for each simplex
    global H, deadsimps
    Rel = getAllRelations(Rel)
    for a in Rel:
        deadsimps.append(a[1])
    for i in range(0, len(Rel)):
        s = MakeLex(Rel[i])
        lor = s[0]
        for u in range(0, len(s)):
            k = h(s[u])%len(H)
            if type(H[k][0]) == list:
                for x in range(0, len(H[k])):
                    if H[k][x][2] == s[u]:
                        lor2 = lowestOrderRelation(s[u])
                        if isLess(lor2, lor):
                            H[k][x][3] = lor2
                            l = h(s[0])%len(H)
                            b = H[l]
                            if type(b[0]) == list:
                                for y in range(0, len(b)):
                                    if b[y][2] == s[0]:
                                        b[y][3] = lor2
                            else:
                                b[3] = lor2
                            lor = lor2
            else:
                H[k][x][3] = lor
        break

```



```

else:
    lor2 = lowestOrderRelation(s[u])
    if isLess(lor2,lor):
        H[k][3] = lor2
        l = h(s[0])%len(H)
        b = H[l]
        if type(b[0]) == list:
            for y in range(0,len(b)):
                if b[y][2] == s[0]:
                    b[y][3] = lor2
        else:
            b[3] = lor2
        lor = lor2
    else:
        H[k][3] = lor
def lowestOrderRelation(s): #finds the lowest order relation of simplex s, using H
    k = h(s)%len(H)
    if type(H[k][0]) == list:
        for i in range(0,len(H[k])):
            if H[k][i][2] == s:
                if H[k][i][3] == s:
                    return s
            else:
                return lowestOrderRelation(H[k][i][3])
    else:
        if H[k][3] == s:
            return s
        else:
            return lowestOrderRelation(H[k][3])
def getAllRelations(Rel):#generates all lower order relations from a relation set Rel
    fullRel = []
    for i in range(0,len(Rel)):
        lowerRels = []
        for u in range(0,len(Rel[i])):
            lowerRels.append(complexify(Rel[i][u],0))
        for k in range(0,len(lowerRels[0])):
            newRel = []
            for x in range(0,len(lowerRels)):
                newRel.append(lowerRels[x][k])
            fullRel.append(newRel)
    return Rel+fullRel
def pruneSimps():
    global D
    d = MakeLex(deadsimps)
    currentdim = 0
    i = 0
    for x in d:
        if len(x)-1>currentdim:

```

```

        currentdim = len(x)-1
        i = 0
        p = location(x)
        D[currentdim].col_del(p - i)
        if currentdim < len(D) - 1:
            D[currentdim + 1].row_del(p - i)
            i = i + 1
def Betti(k,m = None):
    if m == None:
        m = D
    return BettiCalc.Bettislow(k,m)
def ker(k,mat):
    return BettiCalc.ker(k,mat)

```

REFERENCES

- [1] A. Hatcher, *Algebraic Topology*, Cambridge University Press, 2002
- [2] C. Curto, *What can Topology tell us about the neural code?*, Bull. Amer. Math. Soc. 54 (2017), 63-78
- [3] J-G. Dumas et. al., *Computing Simplicial Homology Based on Efficient Smith Normal Form Algorithms*, Algebra Geometry and Software Systems, Springer, 2003, 177-206

CCSU, 1516 STANLEY ST, NEW BRITAIN, CT 06050, AND IMAR, P.O. BOX
 1-764, 014700, BUCHAREST, ROMANIA
E-mail address: marianfantan9@gmail.com

CCSU, 1516 STANLEY ST, NEW BRITAIN, CT 06050
E-mail address: landonrenzullo@gmail.com